Parallel Design and Performance of Nested Filtering Factorization Preconditioner

Long Qu
Université Paris Sud 11
Laboratoire de Recherche en Informatique
Orsay, France
qu@lri.fr

Laura Grigori
INRIA Paris-Rocquencourt, Alpines
and UPMC - Univ Paris 6, CNRS UMR 7598, Laboratoire Jacques-Louis Lions
Paris, France
laura.grigori@inria.fr

Frédéric Nataf
UPMC - Univ Paris 6, CNRS UMR 7598, Laboratoire Jacques-Louis Lions
and INRIA Paris-Rocquencourt, Alpines
Paris, France
nataf@ann.jussieu.fr

ABSTRACT

We present the parallel design and performance of the nested filtering factorization preconditioner (NFF), which can be used for solving linear systems arising from the discretization of a system of PDEs on unstructured grids. NFF has limited memory requirements, and it is based on a two level recursive decomposition that exploits a nested block arrow structure of the input matrix, obtained priorly by using graph partitioning techniques. It also allows to preserve several directions of interest of the input matrix to alleviate the effect of low frequency modes on the convergence of iterative methods. For a boundary value problem with highly heterogeneous coefficients, discretized on three-dimensional grids with 64 millions unknowns and 447 millions nonzero entries, we show experimentally that NFF scales up to 2048 cores of Genci's Bull system (Curie), and it is up to 2.6 times faster than the domain decomposition preconditioner Restricted Additive Schwarz implemented in PETSc.

1. INTRODUCTION

Solving large sparse linear systems of linear equations $Ax = b$ is an operation used in many industrial and academic simulations. Given the complexity of today's numerical simulations and the sizes of the linear systems that need to be solved, the design of linear solvers that have limited memory requirements and that can scale to a large number of processors is of major interest. However, most of the existing methods for solving linear systems, as domain decomposition methods with coarse space corrections or multigrid methods, are not strongly scalable, that is for a given problem size, the numerical efficiency of the preconditioner and/or its parallel performance degrades when the number of processors increases.

This paper focuses on nested filtering factorization (NFF), a multilevel parallel preconditioning technique for solving large sparse linear systems of equations by using iterative methods. NFF is based on a recursive decomposition that requires first to permute the input matrix, which can have an arbitrary sparsity structure, into a matrix with a nested block arrow structure. This recursive factorization is a key feature in allowing NFF to have limited memory requirements and also to be very well suited for hierarchical parallel machines. NFF is also able to preserve some directions of interest of the input matrix A. Given a set of vectors T which represent the directions to be preserved, the preconditioner M satisfies a right filtering property $MT = AT$. This is a property which has been exploited in different contexts, as multigrid methods [6], semiseparable matrices [14], incomplete factorizations [21, 1, 8], or nested factorization [4]. It is well known that for difficult problems with heterogeneities or multiscale physics, the iterative methods can converge very slowly, and this is often due to the presence of several low frequency modes. By preserving the directions corresponding to these low frequency modes in the preconditioner, their effect on the convergence is alleviated and a much faster convergence is often observed. NFF can be seen as an extension of nested factorization [4] that can be used for matrices with arbitrary sparsity structure and for which the computation can be performed in parallel.

The algebra of the recursive decomposition used in NFF has been introduced initially in the technical report [10]. The construction of the preconditioner involves the approximation of Schur complements of the form $LD^{-1}U$ that appear at each step of the recursive decomposition. Different preconditioners can be obtained by using different approximations, and more details can be found in [10]. The specific approximation used in NFF that allows to satisfy the right filtering property was introduced in [12] and is similar to the approach used in block filtering decomposition [11, 8], but adapted to a recursive computation. A multithreaded implementation based on Cilk is reported in [16] and also in [17]. However, in the previous work, the input matrix was reordered by using nested dissection based on bisection and vertex separators, which partitions the graph of the input matrix into two disjoint subgraphs and a separator, and continues recursively this procedure on the two disjoint subgraphs. Typically the recursion depth is $\log P$, where P is the number of processors/compute nodes used in a parallel implementation. When the number of processors is increased, there is an important increase in the arith-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SC '13, November 17 - 21 2013, Denver, CO, USA
Copyright 2013 ACM 978-1-4503-2378-9/13/11...$ 15.00.
http://dx.doi.org/10.1145/2503210.2503287

metic complexity of these preconditioners, which is directly related to the depth of the recursion of the decomposition of NFF, as we explain in more detail later in this paper. This prevented obtaining good performance and scalability in practice for NFF. In this paper, we solve this drawback by using a decomposition of NFF with only two levels of recursion. For this, the input matrix is reordered by using k-way nested dissection with vertex separators, into a matrix with a block arrow structure, in which every diagonal block, except possibly the last one, has itself a block arrow structure. The number of innermost diagonal blocks that can be computed in parallel corresponds to P, the number of processors used in the parallel computation, however the recursion depth is only two. While NFF remains an efficient preconditioner with this new reordering, the other preconditioners presented in [10] require the usage of overlapping techniques as explained in [13].

The main contributions of this paper are the following. We first present the algebra of the NFF preconditioner by using a simpler approach than [10, 12]. We then present the data distribution and the parallel algorithms that allow a very efficient construction and application of the NFF preconditioner. We relate the arithmetic complexity of NFF to the depth of recursion of its decomposition, and with our data distribution and implementation, we estimate its arithmetic and communication complexity. We finally discuss the convergence of NFF on a set of matrices arising from the discretization of a boundary value problem with highly heterogeneous coefficients on three-dimensional grids. Results in section 4 show that on a $400 \times 400 \times 400$ regular grid, the number of iterations with NFF increases slightly while increasing the number of subdomains up to 2048. In terms of runtime performance on Curie, a Bullx system formed by nodes of two eight-core Intel Sandy Bridge processors, NFF scales well up to 2048 cores and it is 2.6 times faster than the domain decomposition preconditioner Restricted Additive Schwarz (RAS) as implemented in PETSc [5]. The choice of the filtering vectors plays an important role in direction preserving preconditioners. There are problems for which we have prior knowledge of the near kernel of the input matrix, and this is indeed the case for the problems tested in this paper. They can also be approximated by using techniques similar to the ones used in deflation, however we do not discuss further this option here.

2. ALGEBRAIC FORMULATION OF NFF

In this section we describe the algebra of NFF preconditioner. It requires the input matrix to have a nested block arrow structure. We briefly recall that such a structure can be obtained by reordering the input matrix based on a nested k-way partitioning with vertex separators of its graph, that we refer to as k-way nested dissection. At the first level, a separator is identified in the graph of the input matrix which disconnects the graph into k disjoint subgraphs. The subgraphs are further partitioned using nested k-way partitioning, and in general the recursion can be stopped at any depth. We refer to the subgraphs obtained at the last level of recursion as subdomains. Note that very often 2-way nested dissection is used, referred to simply as nested dissection. We consider in the rest of the paper that the input matrix was reordered by using two levels of k-way nested dissection. At the first level, its graph was partitioned into q_1 disjoint subgraphs and a separator, while at the second level each subgraph was further partitioned into q_2 disjoint subgraphs and a separator. With this reordering, the permuted matrix, that we refer to as A, has a block arrow structure

$$A = \begin{pmatrix} T_1^1 & & & U_1^1 \\ & T_1^2 & & U_1^2 \\ & & \ddots & \vdots \\ & & T_1^{q_1} & U_1^{q_1} \\ L_1^1 & L_1^2 & \cdots & L_1^{q_1} \quad S_1^1 \end{pmatrix} \tag{1}$$

where $T_1^1, ..., T_1^{q_1}$ correspond to the q_1 disjoint subgraphs and S_1^1 corresponds to the separator obtained from the first level of nested dissection, and $L_1^1, ..., L_1^{q_1}$, $U_1^1, ..., U_1^{q_1}$ represent the interconnection between the subgraphs and the separator. We refer to the matrix formed by all the lower off-diagonal blocks as L_1, and to the matrix formed by all the upper off-diagonal blocks as U_1, that is

$$\begin{aligned} L_1 &= \begin{pmatrix} L_1^1 & L_1^2 & \cdots & L_1^{q_1} \end{pmatrix}, \\ U_1 &= \begin{pmatrix} U_1^{1\,T} & U_1^{2\,T} & \cdots & U_1^{q_1\,T} \end{pmatrix}^T. \end{aligned}$$

We also denote by L_{1ext}, U_{1ext} the matrices of the same size as the original matrix A formed by extending L_1 and U_1 as

$$L_{1ext} = \begin{pmatrix} 0 & 0 \\ L_1 & 0 \end{pmatrix}, \; U_{1ext} = \begin{pmatrix} 0 & U_1 \\ 0 & 0 \end{pmatrix}.$$

Since the diagonal blocks $T_1^1, ..., T_1^{q_1}$ have themselves a block arrow structure, the matrix A can be written as

$$A = \begin{pmatrix} T_2^{1,1} & & & U_2^{1,1} & & & & & & & \\ & \ddots & & \vdots & & & & & & & \\ & & T_2^{1,q_2} & U_2^{1,q_2} & & & & & & & U_1^1 \\ L_2^{1,1} & \cdots & L_2^{1,q_2} & S_2^1 & & & & & & & \\ & & & & \ddots & & & & & & \vdots \\ & & & & & T_2^{q_1,1} & & & U_2^{q_1,1} & & \\ & & & & & & \ddots & & \vdots & & U_1^{q_1} \\ & & & & & & & T_2^{q_1,q_2} & U_2^{q_1,q_2} & & \\ & & & & & L_2^{q_1,1} & \cdots & L_2^{q_1,q_2} & S_2^{q_1} & & \\ & & L_1^1 & & & & \cdots & & L_1^{q_1} & & S_1^1 \end{pmatrix}, \tag{2}$$

where, as an example, the block $T_2^{q_1,1}$ refers to the first diagonal block of $T_1^{q_1}$. Similarly to the previous notation, for $i \in [1..q_1]$, we have

$$\begin{aligned} L_2^i &= \begin{pmatrix} L_2^{i,1} & \cdots & L_2^{i,q_2} \end{pmatrix}, \\ U_2^i &= \begin{pmatrix} U_2^{i,1\,T} & \cdots & U_2^{i,q_2\,T} \end{pmatrix}^T. \end{aligned}$$

The matrices $L_{2,ext}^i$ and $U_{2,ext}^i$ are the matrices of the same size as the diagonal block T_1^i formed by extending L_2^i and U_2^i by blocks of zeros of appropriate dimension.

A separator tree is associated with this partitioning, in which the root of the tree corresponds to the first level separator, and each subtree corresponds to one of the disconnected components. We denote the height of the separator tree obtained from nested dissection as K, which corresponds to the number of recursive calls to k-way graph partitioning with vertex separators. In our case, $K = 2$. The separator tree associated with two levels of 4-way nested dissection is given in Figure 2. In equations (1) and (2), the subscript refers to the level of recursion, and the superscript refers to the number of the separator or the subdomain at

the current level of recursion. Before introducing the preconditioners studied here, we first describe a recursive formulation of an exact factorization of the input matrix A, which was first introduced in [10]. Let A be a matrix with a nested block arrow structure as given in equations (1) and (2). The Schur complement $S_1^{1'}$ can be computed as

$$\begin{aligned} S_1^{1'} &= S_1^1 - L_1 Diag(T_1^1, T_1^2, ..., T_1^{q_1})^{-1} U_1 \\ &= S_1^1 - \sum_{i=1}^{q_1} (L_1^i (T_1^i)^{-1} U_1^i). \end{aligned} \quad (3)$$

where $Diag(...)$ denotes a block diagonal matrix formed by a given set of blocks. The factorization of A can be written as

$$\begin{aligned} A &= (L_{1ext} + F_1) F_1^{-1} (F_1 + U_{1ext}) \quad (4) \\ \text{where} \\ F_1 &= Diag(T_1^1, T_1^2, ..., T_1^{q_1}, S_1^{1'}). \end{aligned}$$

Since each diagonal block T_1^i has itself a block arrow structure as in equation (2), then by computing in addition the Schur complement

$$S_2^{i'} = S_2^i - \sum_{j=1}^{q_2} (L_2^{i,j} (T_2^{i,j})^{-1} U_2^{i,j}), \quad (5)$$

the factorization of T_1^i can be written as

$$\begin{aligned} T_1^i &= (L_{2,ext}^i + F_2^i)(F_2^i)^{-1} (F_2^i + U_{2,ext}^i), \quad (6) \\ \text{where} \\ F_2^i &= Diag(T_2^{i,1}, ..., T_2^{i,q_2}, S_2^{i'}). \end{aligned}$$

Equations (4) and (6) give a recursive factorization of A, which can be continued if the innermost diagonal blocks $T_2^{i,j}$ have themselves a block arrow structure. In this exact recursive factorization, the computation of the Schur complements from equations (3) and (5) is expensive in terms of both computation and memory requirements. The idea behind NFF preconditioner (that we refer to as B_{NFF}) is to approximate the Schur complements by sparse matrices such that the computation as well as the application of the preconditioner can be performed efficiently. In addition, this approximation is performed such that NFF preserves directions of interest from the matrix A. In this paper we focus on the case when only one direction of interest is preserved, that is given a filtering vector t, the right filtering condition $B_{NFF} t = At$ is satisfied. However the preconditioner can be extended to preserve multiple directions of interest.

At the first level of recursion, the preconditioner NFF is given by the factorization

$$B_{NFF} = (L_{1,ext} + \widetilde{F}_1) \widetilde{F}_1^{-1} (\widetilde{F}_1 + U_{1,ext}), \quad (7)$$

where

$$\widetilde{F}_1 = \begin{pmatrix} \widetilde{T}_1^1 & & & \\ & \ddots & & \\ & & \widetilde{T}_1^{q_1} & \\ & & & S_1^1 - \sum_{i=1}^{q_1} L_1^i \beta_1^i U_1^i \end{pmatrix}. \quad (8)$$

In this factorization, $\widetilde{F}_1$ is a matrix which approximates the block diagonal matrix F_1 of the exact factorization of the matrix A from equation (4). In more detail, $\widetilde{F}_1$ is computed by approximating each diagonal block T_1^i by a matrix $\widetilde{T}_1^i$, and by approximating each matrix $(T_1^i)^{-1}$ involved in the Schur complement computation corresponding to the last diagonal block of $\widetilde{F}_1$ by a matrix β_1^i, such that the filtering condition is satisfied. Let the filtering vector t be partitioned as $t^T = [t_1^{1\,T} ... t_1^{q_1\,T} t_1^{s\,T}]$ by following the same partition as the matrix A in equation (1). The filtering condition is satisfied by B_{NFF} if every approximation performed during its construction satisfies as well a corresponding filtering property, that is

$$\widetilde{T}_1^i t_1^i = T_1^i t_1^i \ , \beta_1^i U_1^i t_1^s = (\widetilde{T}_1^i)^{-1} U_1^i t_1^s, \ \forall i \in [1..q_1]. \quad (9)$$

Since the blocks $\widetilde{T}_1^1, ..., \widetilde{T}_1^{q_1}$ have themselves a block arrow structure, they are factored by using the same approach as in equation (7). The innermost diagonal blocks obtained from their factorization should be sufficiently small, such that they can be factored by using an exact LU factorization. Once the diagonal blocks $\widetilde{T}_1^i$, $i \in [1..q_1]$ are factored, the approximation matrices β_1^i , $i \in [1..q_1]$ are computed. The last diagonal block of $\widetilde{F}_1$ can be then calculated and afterwards factored. We explain later the construction of a diagonal matrix β_1^i which respects the condition in equation (9). The factorization of NFF for a matrix reordered by using two levels of k-way nested dissection is presented in Definition 1.

Definition 1. Let A be an $n \times n$ matrix which has a nested block arrow structure as given in equations (1) and (2), obtained by using two levels of k-way nested dissection. Given a filtering vector t of size n, let

$$\begin{aligned} \widetilde{S}_2^i &= S_2^i - \sum_{j=1}^{q_2} L_2^{i,j} \beta_2^{i,j} U_2^{i,j}, \forall i \in [1..q_1]. \\ \widetilde{F}_2^i &= Diag(T_2^{i,1}, ..., T_2^{i,q_2}, \widetilde{S}_2^i), \forall i \in [1..q_1], \\ \widetilde{T}_1^i &= (L_{2,ext}^i + \widetilde{F}_2^i)(\widetilde{F}_2^i)^{-1} (\widetilde{F}_2^i + U_{2,ext}^i), \forall i \in [1..q_1], \\ \widetilde{S}_1^1 &= S_1^1 - \sum_{i=1}^{q_1} L_1^i \beta_1^i U_1^i, \\ \widetilde{F}_1 &= Diag(\widetilde{T}_1^1, ..., \widetilde{T}_1^{q_1}, \widetilde{S}_1^1), \end{aligned}$$

where we suppose that the matrices $T_2^{i,j}$ for all $i \in [1..q_1]$, $j \in [1..q_2]$, $\widetilde{F}_2^i$ for all $i \in [1..q_1]$, and $\widetilde{F}_1$ are invertible. Let the filtering vector be partitioned as $t^T = [t_1^{1\,T} ... t_1^{q_1\,T} t_1^{s\,T}]$ by following the same partitioning of A in equation (1). Further let each subvector t_1^i be partitioned as $t_1^{i\,T} = [t_2^{i,1\,T} ... t_2^{i,q_2\,T} t_2^{i,s\,T}]$ by following the same partitioning of the block T_1^i in equation (2). The matrices β_1^i and $\beta_2^{i,j}$, $i \in [1..q_1], j \in [1..q_2]$ are approximation matrices chosen such that they satisfy,

$$\begin{aligned} \beta_1^i U_1^i t_1^s &= (\widetilde{T}_1^i)^{-1} U_1^i t_1^s, \forall i \in [1..q_1], \quad (10) \\ \beta_2^{i,j} U_2^{i,j} t_2^{i,s} &= (T_2^{i,j})^{-1} U_2^{i,j} t_2^{i,s}, \forall i \in [1..q_1], \forall j \in [1..q_2]. \end{aligned}$$

The NFF preconditioner B_{NFF} of A is defined as

$$\begin{aligned} B_{NFF} &= \widetilde{T}_0^1 \\ &= (L_{1,ext} + \widetilde{F}_1)(\widetilde{F}_1)^{-1} (\widetilde{F}_1 + U_{1,ext}). \end{aligned}$$

When $U_1^i t_1^s$ is a vector of nonzero elements, a diagonal matrix β_1^i that satisfies equation (10) can be computed as $diag(((\widetilde{T}_1^i)^{-1} U_1^i t_1^s)./(U_1^i t_1^s))$, where ./ denotes pointwise division and $diag(...)$ here refers to a matrix whose diagonal is formed from a given vector. A slightly different and more

general construction can be used [11], which works also when $U_1^i t_1^s$ has zero elements. $\beta_2^{i,j}$ can be computed in a same way.

A different preconditioner introduced in [10] is NSSOR. While we do not give its detailed formulation, we just mention that NSSOR is defined similarly to NFF, except that in NSSOR, $\widetilde{F}_2^i = Diag(T_2^{i,1}, ..., T_2^{i,q_2}, S_2^i)$ and $\widetilde{F}_1 = Diag(\widetilde{T}_1^1, ..., \widetilde{T}_1^{q_1}, S_1^1)$, that is all the Schur complements are dropped in the construction of the preconditioner. This leads to a preconditioner with a very cheap construction cost, since it requires only the factorization of diagonal blocks. In our recent work [13], we have shown that by using overlapping techniques similar to the ones from domain decomposition methods, the convergence of NSSOR is greatly improved. Without going further into the details of the overlapping techniques, we note that the experiments from section 4 present the convergence of NSSOR with overlapping, in which the variables at the interface between the different subdomains are duplicated.

While the construction of NSSOR and NFF is different, their application to a vector during the iterative process is identical and involves recursive calls to lower levels of the preconditioner. We explain in more detail the construction and the application of the preconditioners in the following section, directly in a parallel context.

3. PARALLEL NFF

In this section we describe the parallel algorithms and the data distribution that allow an efficient construction and application of NFF, for matrices having two levels of nested block arrow structure. NFF can be used with different Krylov subspace methods, and in our work we use the built-in GMRES iterative solver [19] from PETSc [5].

Reordering and relation with arithmetic complexity.

When NFF is executed on P processors, the input matrix is reordered using a two-level k-way nested dissection with q_1 partitions in the upper level and q_2 partitions in the lower level, such that $q_1 \cdot q_2 = P$. In the following we refer to the P processors as $P_1 \dots P_P$. There are two reasons for which we use only two levels of recursion in the graph partitioning reordering. The first reason is the arithmetic complexity. Applying NFF in the iterative solver (the algorithm is given later in this section) relies on calling recursively solves that involve separators and subdomains. More precisely, for K levels, the iterative solver with NFF requires one solve with the first level separator $\widetilde{S}_1^1$, two solves with each second level separator $\widetilde{S}_2^i$, and the computation cost continues to increase. In general there are 2^{j-1} solves with each separator at the j-th level and 2^K solves with each subdomain. Suppose that the solves associated with the subdomains dominate the overall computation. If 2-way nested dissection with a recursion depth of $\log P$ is used, then adding one extra level in the separator tree to double the number of processors used and the number of subdomains results in roughly doubling the amount of computation required for applying NFF. In this case, there is no benefit obtained from increasing the number of processors, and this prevents obtaining a scalable preconditioner. To limit the arithmetic complexity, we use only two levels of k-way nested dissection. The second reason is that such a structure is well adapted for hierarchical machines with two levels of parallelism as for example clusters of multicore processors. The preconditioner can be computed and applied in parallel by using the appropriate level of parallelism at a given level of computation.

Data distribution.

The most common distribution used in iterative solvers is the block row distribution, in which a contiguous block of rows is assigned to one processor for both the input matrix A and the vectors associated with the iterative solver. This is indeed the data distribution used in PETSc for GMRES. We use this distribution for the input matrix A, and the parallel matrix-vector product of the form Ab performed at each iteration of GMRES will rely on it. However this distribution is not appropriate for our preconditioner NFF. The solve process with NFF requires not only solves in local subdomains, but also several matrix-vector multiplications of the form $L_i^j x^j$ and $U_i^j x^i$, where L_i^j, U_i^j are submatrices of A and x^i, x^j are vectors of appropriate dimensions. With a block row distribution, L_i^j and x^j are not always stored on a same processor, and performing the matrix-vector multiplication will require communication, that can be avoided by different distributions.

NFF uses an arrow distribution and a subtree to subcube mapping [9] of data to processors. We define the following sets of processors,

$$\begin{aligned}
\Pi_1^1 &= \{P_1, P_2, ..., P_{q_1 \cdot q_2}\}, \\
\Pi_2^i &= \{P_{1+(i-1)q_2}, P_{2+(i-1)q_2}, ..., P_{i \cdot q_2}\}, \forall i \in [1...q_1], \\
\Pi_3^i &= \{P_i\}, \forall i \in [1...q_1 \cdot q_2].
\end{aligned}$$

We describe now the data distribution by using a bottom-up traversal of the separator tree. Each subdomain situated at the leaves of the separator tree and corresponding to the diagonal block $T_2^{i,j}$, is stored on the processor $P_{(i-1)q_2+j}$ (the set $\Pi_3^{(i-i)q_2+j}$). In addition, the block column of L and the block row of U corresponding to the interconnection of this subdomain with the separators are stored on the same processor $P_{(i-1)q_2+j}$. Each i-th lower level separator, which corresponds to the diagonal block S_2^i, is assigned to the set of processors Π_2^i. The set Π_2^i is formed by all the processors assigned to the leaves of the subtree for which S_2^i is the root. Similarly, using an arrow distribution, the corresponding block column of L and block row of U are assigned to the same set of processors. And finally the upper level sep-

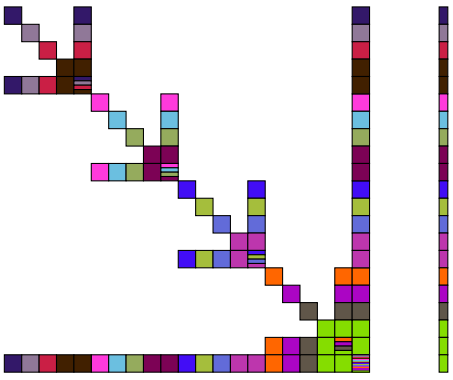

Figure 1: Data distribution of a matrix with a two-level block arrow structure and a vector involved in the iterative process.

arator corresponding to the diagonal block is assigned to all the processors, the set Π_1^1 with our notation.

With this mapping, the computation that involves a subdomain or its interconnections with the upper levels separators is performed on only one processor. The computation that involves lower level separators and interconnections with the upper level separator is performed in parallel on disjoint sets of processors and the communication is restricted only within a set of processors. And finally all the processors participate to the computation of the last level separator. This data distribution allows to reduce communication during the computation of the Schur complements and when multiplying blocks of L and U with distributed vectors. It is illustrated in Figure 1 for a matrix with a nested block arrow structure, with 4 diagonal blocks and a separator block at each level, which is distributed over 16 processors (that correspond to 16 MPI processes). We distinguish the different processors by using different colors. The set of processors assigned to each node in the separator tree is displayed in Figure 2. We use a similar distribution for the right hand side b and the solution vector as displayed in Figure 1.

In the following, given a distributed vector v, we refer to the subvector of v stored on the processors in the set Π_i^j as $v(\Pi_i^j)$. For a distributed matrix M, we refer to the submatrix of M stored on the processors in the set Π_i^j as $M(\Pi_i^j)$. When the set Π_i^j is formed by only one processor P_i, we use the notation $v(P_i)$ and $M(P_i)$. Each block of the factors L, F, and U of B_{NFF} is stored in Compressed Sparse Row format (CSR), a format supported by SparseBLAS [7] and most of the available sparse direct solvers.

Construction of the preconditioner.

The recursive construction of NFF is described in Algorithm 1. The first call to this algorithm is of the form `ComputeNFF(A,1,K,`Π_1^1`,t)`, where A is the input matrix distributed as described previously, $K = 2$ is the height of the separator tree, and t is the filtering vector. First, an approximate factorization $\widetilde{T}_k^i$ is computed for each diagonal block T_k^i, $i = 1, \ldots, q$, through a recursive call to `ComputeNFF`. These approximate factorizations are performed in parallel by using disjoint sets of processors. Once they are completed, the computation of the last diagonal block $\widetilde{S}_k^1 = S_k^1 - \sum_{i=1}^q L_k^i \beta_k^i U_k^i$ can be performed. At the first level of the recursion, this computation reflects the algebra presented in equations (7) and (8) from section 2. The computation of the last diagonal block is performed as follows. For each diagonal block $\widetilde{T}_k^i$ we compute first an approximation of its inverse $\beta_k^i = diag(((\widetilde{T}_k^i)^{-1} U_k^i t(\Pi_k^i))./(U_k^i t(\Pi_k^i)))$, where $t(\Pi_k^i)$ is the part of the filtering vector t stored on the set of processors Π_k^i. We first compute $u_i = U_k^i t(\Pi_k^i)$ by using all the processors in the set Π_k^i. This can be computed

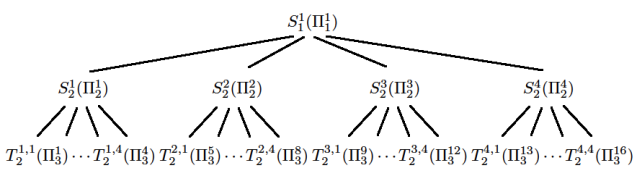

Figure 2: Subtree to subcube mapping for the example matrix from Figure 1.

Algorithm 1 ComputeNFF (T, k, K, Π_k, t): compute B_{NFF} preconditioner of a given matrix T.

Input: K is the recursion depth, k is the current level of recursion, Π_k is the processor set, t is the filtering vector distributed on Π_k as described in Figure 1.
If $k \leq K$, then T has the structure

$$T = \begin{pmatrix} T_k^1 & & & & U_k^1 \\ & T_k^2 & & & U_k^2 \\ & & \ddots & & \vdots \\ & & & T_k^q & U_k^q \\ L_k^1 & L_k^2 & \cdots & L_k^q & S_k^1 \end{pmatrix}$$

where q is the number of partitions at the current level k. For each $1 \leq i \leq q$, L_k^i, T_k^i, and U_k^i are located on the same subset of processors Π_{k+1}^i. S_k^1 is located on the last processor in the set Π_k.
Output: Updated factored form of diagonal blocks, approximate factorization $\widetilde{T}$ of T
if $k > K$ **then**
  $\widetilde{T} = T$, factor $\widetilde{T}$ using the processors in Π_k
else
  for each i from 1 **to** q in parallel **do**
    $\widetilde{T}_k^i \leftarrow$ ComputeNFF$(T_k^i, k+1, K, \Pi_{k+1}^i, t)$
    Each processor $j \in \Pi_{k+1}^i$ computes $u_i(j) = U_k^i(j) t(j)$
    $v_i \leftarrow$ SolveParallel $(T_k^i, u_i, k+1, K, \Pi_{k+1}^i)$
    Each processor $j \in \Pi_{k+1}^i$ computes
      $\beta_k^i(j) = u_i(j)./v_i(j)$
    Each processor $j \in \Pi_{k+1}^i$ computes
      $C^j = L_k^i(j) \beta_k^i(j) U_k^i(j)$
  end for
  Let P_{last} be the last processor in the set Π_k
  if my processor is P_{last} **then**
    Recv C^j from each processor j in Π_k
    Compute $\widetilde{S}_k^1 = S_k^1 - \sum_{\forall j \in \Pi_k^i} C^j$
  else
    Send C^j to P_{last}, where j is my processor number
  end if
  Factor $\widetilde{S}_k^1$ in parallel using all processors in Π_k
  Let $\widetilde{T}$ be formed as

$$\begin{pmatrix} \widetilde{T}_k^1 & & & \\ & \ddots & & \\ & & \widetilde{T}_k^n & \\ L_k^1 & \cdots & L_k^q & \widetilde{S}_k^1 \end{pmatrix} \begin{pmatrix} \widetilde{T}_k^1 & & & \\ & \ddots & & \\ & & \widetilde{T}_k^q & \\ & & & \widetilde{S}_k^1 \end{pmatrix}^{-1} \begin{pmatrix} \widetilde{T}_k^1 & & & U_k^1 \\ & \ddots & & \vdots \\ & & \widetilde{T}_k^q & U_k^q \\ & & & \widetilde{S}_k^1 \end{pmatrix}$$

end if

with no communication since with the arrow distribution, each processor j in the set Π_k^i holds its part of the filtering vector t and a block row of U_k^i. This results in a vector u_i distributed over the processors in the set Π_k^i. This vector will be used as a right hand side for an inner parallel solve $\widetilde{T}_k^i v_i = u_i$ performed by calling `SolveParallel`. The algorithm of this routine, which relies on the factored form of the inner blocks $\widetilde{T}_{k+1}^j$ and $\widetilde{S}_{k+1}^1$, will be discussed later in this section. We mention that only two communication stages are required during this solve. Once the solve is done, we obtain another distributed vector v_i. Afterwards, each processor j in the subset Π_{k+1}^i computes a local part of β_k^i denoted by $\beta_k^i(j)$ based on its local vectors $u_i(j)$ and $v_i(j)$ and then it computes a local part of the Schur complement $C^j = L_k^i(j) \beta^i(j) U_k^i(j)$. After a communication step, the last processor in the set Π_k^i accumulates from all the other

processors j in the set the local matrices C^j and updates the last diagonal block S_k^1. Then all the processors in the set Π_k compute the LU factorization of S_k^1.

At the last level of recursion, the LU factorization of the subdomains is computed by using one processor per subdomain. We note that in practice a multi-threaded sparse direct solver can be used to solve subdomains locally and a distributed sparse direct solver can be used to solve separators in parallel.

For all these algorithms, the approximation matrix $\widetilde{T}$ is never computed explicitly. The only blocks modified are the innermost diagonal blocks corresponding to the subdomains and to the separators. While we do not give a detailed complexity analysis, we conclude that B_{NFF} is efficient in terms of memory consumption, since it requires to store only the LU factors of the innermost diagonal blocks and the separators.

Application of the preconditioner.

At each iteration of the Krylov subspace solver, applying B_{NFF} requires solving $B_{NFF}x = b$, where b is a given right hand side and x is the solution sought. It can be written as

$$\begin{aligned} B_{NFF} &= \widetilde{T}_0^1 \\ &= (L_{1,ext} + \widetilde{F}_1)(\widetilde{F}_1)^{-1}(\widetilde{F}_1 + U_{1,ext}) \\ &= (\underbrace{(L_{1,ext} + \widetilde{F}_1)}_{\widetilde{T}_L} \cdot (\underbrace{I + \widetilde{F}_1^{-1}U_{1,ext}}_{\widetilde{T}_U}), \end{aligned}$$

and can be solved through a forward-backward solve,

$$\begin{aligned} \widetilde{T}_L y &= b, \\ \widetilde{T}_U x &= y. \end{aligned}$$

Therefore, the solving process described in Algorithm 2 is composed of a call to `ForwardSolve`$(\widetilde{T}_L, b, k, K, \Pi_1^1)$ and a call to `BackwardSolve` $(\widetilde{T}_U, y, k, K, \Pi_1^1)$, presented in Algorithms 3 and 4 respectively. Based on the recursive definition of B_{NFF}, both algorithms `ForwardSolve` and `BackwardSolve` are recursive. We describe in the following the `ForwardSolve` algorithm, and we note that the `BackwardSolve` is based on a very similar approach. The forward solve needs to solve the equation $\widetilde{T}_L y = b$, and equation (11) gives the structure of $\widetilde{T}_L$ and the corresponding partitioning of the vectors y and b. The forward solve starts by computing in parallel $\widetilde{T}_k^i y^i = b^i$ through a recursive call to `SolveParallel` which exploits the recursive factorization of the diagonal block $\widetilde{T}_k^i$, where $i = 1, \ldots, q$.

Once the vectors $y^1, \ldots, y^q$ are computed, the algorithm needs to compute $\widetilde{S}_k^1 y^s = b^s - \sum_{i=1}^q L_k^i y^i$. Due to the arrow distribution, the matrix vector multiplication $L_k^i(j)y^i(j)$ can be performed locally on each processor j, followed by a reduce operation to compute the summation $\sum_{i=1}^q \sum_{\forall j \in \Pi_{k+1}^i} L_k^i(j)y^i(j)$. Since processor sets Π_{k+1}^i are disjoint, each $j \in \Pi_{k+1}^i$ is unique. Therefore, we can note $L_k^i(j)y^i(j)$ by s^j. Thus, the summation can be simplified as $\sum_{\forall j \in \Pi_k} s^j$.

As a result, `ForwardSolve` requires only one collective communication based on `MPI_Reduce` to compute y^s, while `BackwardSolve` requires one collective communication based on `MPI_Bcast` to redistribute y^s to all the processors in the set Π_k. We note that the same solver which was used for computing the factorization of the diagonal blocks of the subdomains or the separators is also used for the solves. Figure 3 shows the solving process workflow for one level 4-way nested dissection based on algorithms described before. For each processor, it requires two local solves, one parallel solve, two local sparse matrix vector multiplications, one `MPI_Reduce`, one `MPI_Bcast`, and one local subtraction.

Complexity of the solving process.

We discuss the complexity in terms of flops and communication of the solving process of NFF. Since this depends on many factors, as the structure of the input matrix, the number of subdomains, we consider here model problems discretized on 3D regular grids. Suppose that we have a 3D regular grid, the associated matrix is of size $n \times n$, and its graph is partitioned by using a two-level k-way partitioning, with q_1 separators in the lower level and $q_1 \cdot q_2$ subdomains,

---

Algorithm 2 SolveParallel $(\widetilde{T}, b, k, K, \Pi_k)$: solve $\widetilde{T}x = b$ in parallel

Input: K is total recursion depth, k is the current level of recursion. Π_k is the processor set. The procedure solves the system $\widetilde{T}x = b$ with $\widetilde{T} = \widetilde{T}_L\widetilde{T}_U$
Output: x
Solve $\widetilde{T}_L y = b$ by calling ForwardSolve$(\widetilde{T}_L, b, k, K, \Pi_k)$
Solve $\widetilde{T}_U x = y$ by calling BackwardSolve$(\widetilde{T}_U, y, k, K, \Pi_k)$

Algorithm 3 ForwardSolve$(\widetilde{T}_L, b, k, K, \Pi_k)$: solve $\widetilde{T}_L y = b$ in parallel

Input: K is total recursion depth, k is the current level of recursion. Π_k is the set of processors used. If $k \leq K$, then the procedure solves the system $\widetilde{T}_L y = b$, that is

$$\underbrace{\begin{pmatrix} \widetilde{T}_k^1 & & & \\ & \ddots & & \\ & & \widetilde{T}_k^q & \\ L_k^1 & \cdots & L_k^q & \widetilde{S}_k^1 \end{pmatrix}}_{\widetilde{T}_L} \cdot \underbrace{\begin{pmatrix} y^1 \\ \vdots \\ y^q \\ y^s \end{pmatrix}}_{y} = \underbrace{\begin{pmatrix} b^1 \\ \vdots \\ b^q \\ b^s \end{pmatrix}}_{b}. \qquad (11)$$

where b is the given right hand side and q is the partition number of current level k. For each $1 \leq i \leq q$, L_k^i,T_k^i, U_k^i, b_i, and x_i are located on the same processor subset Π_{k+1}^i. $\widetilde{S}_k^1$ is located on the last processor in the set Π_k referred to as P_{last}.
Output: vector y
if $k > K$ **then**
  Solve $\widetilde{T}_L y = b$
else
  for each i from 1 **to** q in parallel **do**
    $y^i \leftarrow$SolveParallel$(\widetilde{T}_k^i, b^i, k+1, K, \Pi_{k+1}^i)$
    Each processor $j \in \Pi_{k+1}^i$ computes $s^j = L_k^i(j)y^i(j)$
  end for
  Call `MPI_Reduce` with s^j where j is my processor number, and store $\sum_{\forall j \in \Pi_k} s^j$ on P_{last}
  if my processor is P_{last} **then**
    Compute $b^s - \sum_{\forall j \in \Pi_k} s^j$
  end if
  Solve $\widetilde{S}_k^1 y^s = b^s - \sum_{i=1}^q s^i$ in parallel by using all processors in Π_k
end if

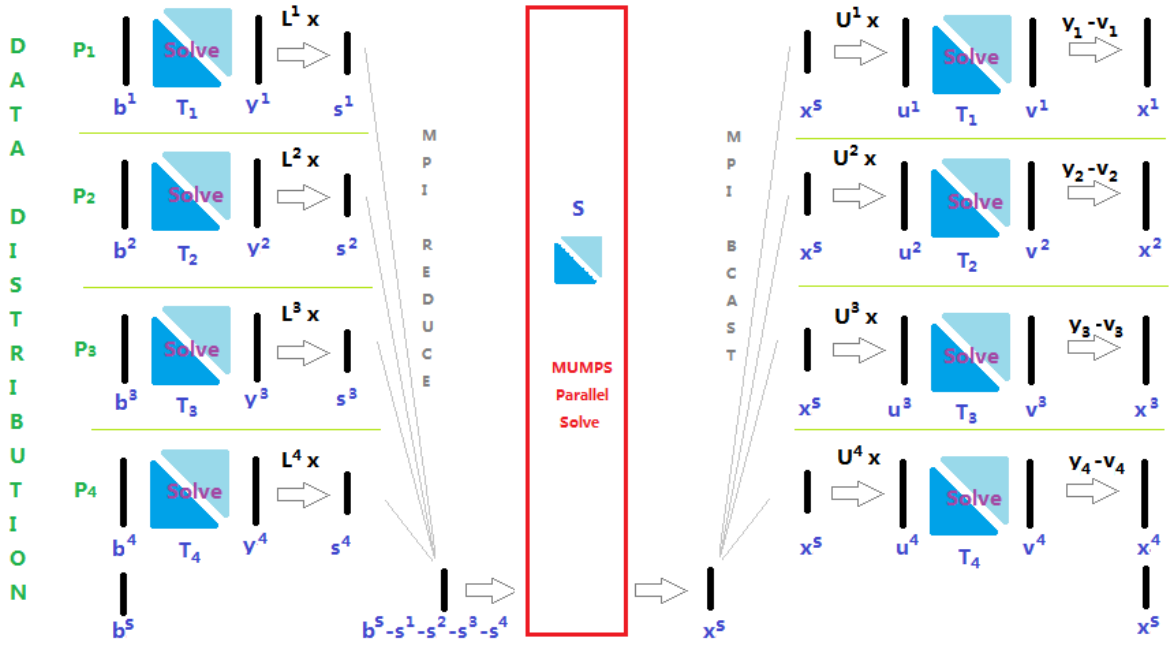

Figure 3: Workflow of the solving process for one-level 4-way nested dissection.

Algorithm 4 BackwardSolve($\widetilde{T}_U, y, k, K, \Pi_k$): solve $\widetilde{T}_U x = y$ in parallel

Input: K is total recursion depth, k is the current level of recursion, Π_k is the processor set. If $k \leq K$, then the procedure solves the system $\widetilde{T}_U x = y$, that is

$$\underbrace{\begin{pmatrix} I & & (\widetilde{T}_k^1)^{-1}U_k^1 \\ & \ddots & \vdots \\ & I & (\widetilde{T}_k^q)^{-1}U_k^q \\ & & I \end{pmatrix}}_{\widetilde{T}_U} \cdot \underbrace{\begin{pmatrix} x^1 \\ \vdots \\ x^q \\ x^s \end{pmatrix}}_{x} = \underbrace{\begin{pmatrix} y^1 \\ \vdots \\ y^q \\ y^s \end{pmatrix}}_{y}.$$

where y is the given right hand side and q is the partition number of current level k. For each $1 \leq i \leq q$, L_k^i,T_k^i, U_k^i, b_i and x_i are located on the same processor subset Π_{k+1}^i. $\widetilde{S}_k^1$ is stored on P_{last}, the last processor in the set Π_k.

Output: vector x

if $k > K$ **then**
  Solve $\widetilde{T}_U x = y$
else
  Set $x^s \leftarrow y^s$ in P_{last}
  Call `MPI_Bcast` to distribute x^s from P_{last} to all processors in Π_k
  for each i from 1 **to** q in parallel **do**
    Each processor $j \in \Pi_{k+1}^i$ computes $u^i(j) = U_k^i(j)x^s$
    $v^i \leftarrow$ SolveParallel($\widetilde{T}^i, u^i, k+1, K, \Pi_{k+1}^i$)
    Each processor $j \in \Pi_{k+1}^i$ computes $x^i(j) = y^i(j) - v^i(j)$
  end for
end if

$P = q_1 \cdot q_2$ is the total number of processors used in the algorithm. To simplify the analysis, we consider the case where q_1 and q_2 are both a power of 8, such that during the recursive partitioning, the domain will be divided into q_i, $i = 1, 2$ regular cubic subdomains. The total computational cost on the critical path is

$$\begin{aligned} \#flops \quad \approx \quad & 4 \max_{i=1..q_1, j=1..q_2} sol(T_2^{i,j}) + 2 \max_{i=1..q_1} sol(S_2^i) \\ & + sol(S_1^1) + 2(\frac{24}{q_2} + 3\log q_2)(\sqrt[3]{q_2} - 1)\frac{\sqrt[3]{n^2}}{\sqrt[3]{q_1}} \\ & + (\frac{24}{P} + 3\log P)(\sqrt[3]{q_1} - 1)\sqrt[3]{n^2} + 4\frac{n}{P}, \end{aligned}$$

where $sol(T_2^{i,j})$, $sol(S_2^i)$ and $sol(S_1^1)$ are the computational costs of the backward-forward solves of the solvers used for the innermost diagonal blocks of the preconditioner. Here $T_2^{i,j}$ refers to the $((i-1)q_2 + j)^{th}$ subdomain, S_2^i is the i^{th} lower level separator and S_1^1 is the upper level separator.

The total communication cost performed on the critical path in terms of volume of communication and number of messages is

$$\begin{aligned} \#words \quad = \quad & 12(\sqrt[3]{q_2} - 1)\frac{\sqrt[3]{n^2}}{\sqrt[3]{q_1}}\log q_2 + 2comw(S_1^1) \\ & + 6(\sqrt[3]{q_1} - 1)\sqrt[3]{n^2}\log P + \max_{i=1...q_1} comw(S_2^i), \\ \#messages \quad = \quad & 2\log P + 4\log q_2 \\ & + 2comm(S_1^1) + \max_{i=1...q_1} comm(S_2^i), \end{aligned}$$

where $comw(S_1^1)$ and $comw(S_2^i)$ refer to the volume of communication required for the solve in the upper level separator S_1^1 and the i-th lower level separator respectively. Similarly, $comm(S_1^1)$ and $comm(S_2^i)$ refer to the number of messages. As the cost of factorization of a block depends on its size, when the number of partitions q_1 and q_2 increases, the computational cost in each processor decreases. However, the communication cost increases.

4. NUMERICAL RESULTS

In this section we discuss the numerical efficiency and the performance on parallel machines of NFF, and compare it with Restricted Additive Schwarz (RAS) as implemented in PETSc, a widely used domain decomposition method. Our tests are performed on the Curie supercomputer[1], based on Bullx system. Curie is composed of 5040 nodes, each node is composed of two eight-core Intel Sandy Bridge processors (2.7GHz). InfiniBand QDR Full Fat Tree is the high performance interconnect used for the network. Our codes are compiled with Intel compilers and MKL version 13.1.3.192, BullxMPI version 1.1.16.5, MUMPS version 4.10.0, and PETSc version 3.4.2. In our implementation, for both NFF and NSSOR, the input matrix is reordered into a nested block arrow matrix by using the nested dissection routine from Metis [15]. We perform $\log(P)$ levels of nested dissection to obtain P independent subdomains, and then we group together the separators to obtain the desired separator tree of height two. We refer to the graph partitioning technique as $q_2 \times q_1$ nested dissection, which reflects the nested block arrow structure given in equations (1) and (2). For RAS we use natural ordering and we divide the $n \times n$ input matrix into P subdomains, each of size n/P (except possibly the last block if n is not a multiple of P). This is feasible since the test matrices arise from discretization on 3D regular grids. In a more general setting, k-way partitioning with edge separators should be used. For the factorization of the innermost diagonal blocks, we test the usage of both PARDISO [20] and MUMPS [3].

Our test matrices are obtained from the discretization of the Poisson problem and the convection-diffusion problem on 3D regular grids. The 3D convection-diffusion problem was also considered in previous papers on filtering and nested preconditioners, see for example [1, 2, 18, 10]. They are all related to the following boundary value problem,

$$\begin{aligned} div(\mathbf{a}(x)u) - div(\kappa(x)\nabla u) &= f \ in \ \Omega \\ u &= 0 \ on \ \partial\Omega_D \\ \frac{\partial u}{\partial n} &= 0 \ on \ \partial\Omega_N \end{aligned}$$

where $\Omega = [0,1]^3$, $\partial\Omega_N = \partial\Omega \setminus \partial\Omega_D$, and $\partial\Omega_D = [0,1] \times \{0,1\} \times [0,1]$. The vector field $\mathbf{a}$ and the tensor κ are the given coefficients of the partial differential operator. As there are several variants of this problem, we choose Skyscraper as an example, where $\mathbf{a}(x) = 0$ and the tensor κ is isotropic and discontinuous,

$$\kappa(x) = \begin{cases} 10^3([10x_2]+1), if \ [10x_i] \text{ is odd}, i = 1,2,3, \\ 1, \qquad\qquad\qquad\qquad otherwise. \end{cases}$$

With this configuration, the domain contains many disjoint zones of high permeability. As shown in Figure 4, the discontinuous κ forms a grid of cubes in the space, where each cube represents a zone of high permeability. In the following, we denote the matrix from the Poisson problem ($\kappa \equiv 1$) as `3DPOImxmxm` and the Skyscraper variant of the boundary value problem as `3DSKYmxmxm`, where m corresponds to the grid size. Since we use 7-point stencils in 3D, the matrix size is $n = m^3$ and the number of nonzero entries is approximately equal to $7m^3$.

Choosing the filtering vector t is a key part of NFF. In our tests, we use $\mathbb{1}$, a vector with m^3 entries all equal to 1. The main reason of this choice is that our test matrices have a row sum equal to 0 in the interior points of the grid. The vector $\mathbb{1}$ is close to an eigenvector corresponding to an eigenvalue near zero. Moreover, there are simulations for which filtering on $\mathbb{1}$ ensures preserving some physical properties. As explained in [4], in a reservoir simulation this ensures a mass conservation property. Figure 5 displays all the eigenvalues of `3DSKY15x15x15` as well as `3DSKY15x15x15` preconditioned by NFF. In this experiment, the matrix is reordered using 2×2 nested dissection. As it can be seen, most of the eigenvalues of the preconditioned `3DSKY15x15x15` with NFF are clustered around 1. This indicates that NFF is efficient in improving the condition number of the input matrix, and a good convergence rate can be expected. (We use a small matrix to be able to compute all its eigenvalues.)

We use `3DPOI225x225x225` and `3DSKY225x225x225` to evaluate the convergence rate of NFF. They have around 11 million unknowns and 70 million nonzero entries. In these tests, we use a random right-hand side with values from -5 to 5 and GMRES which restarts at 200 iterations and stops if the relative residual is smaller than 10^{-8}. We compare its convergence rate with that of Restricted Additive Schwarz (RAS) and NSSOR with overlapping. Figures 6 and 8 display the evolution of the relative residual obtained with GMRES preconditioned with RAS (red), NSSOR (blue) and NFF (green) with the number of subdomains varying from 64 to 2048. The number of partitions q_1, q_2 used at each level of the recursion is a power of 2 and they are as close as possible to each other. For example, we use 8×8 partitions for 64 subdomains and 16×8 partitions for 128 subdomains. We observe that for all these tests, NFF has the

[1] http://www-hpc.cea.fr/en/complexe/tgcc-curie.htm

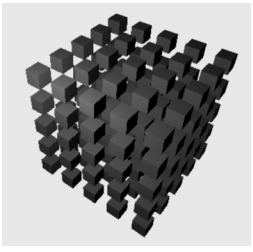

Figure 4: κ in 3D Skyscraper problems where cubes in the space represent zones of high magnitude values.

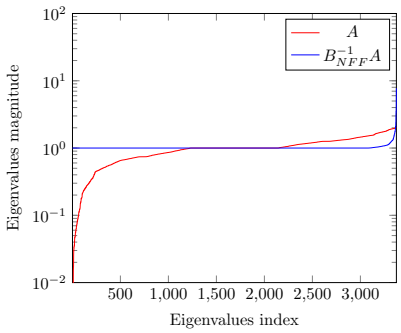

Figure 5: Spectrum of matrix `3DSKY15x15x15`.

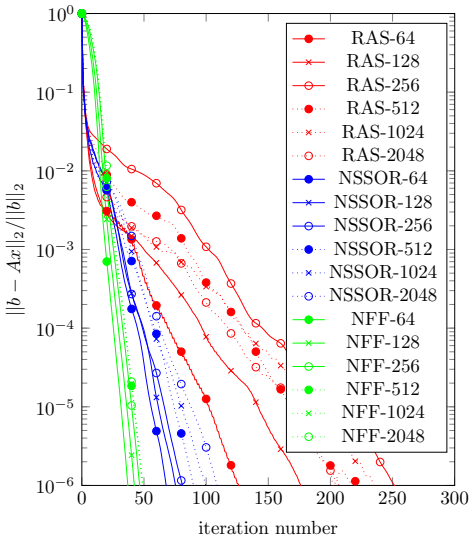

Figure 6: Convergence for `3DPOI225x225x225`.

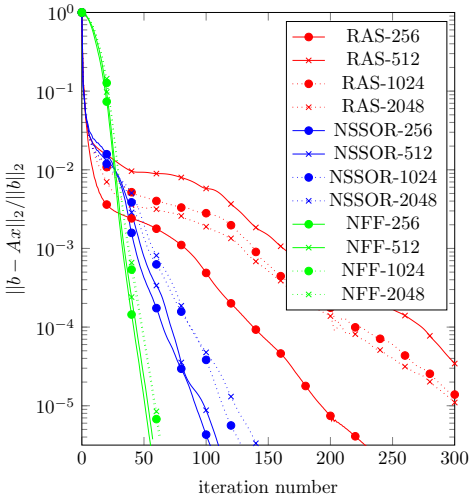

Figure 7: Convergence for `3DPOI400x400x400`.

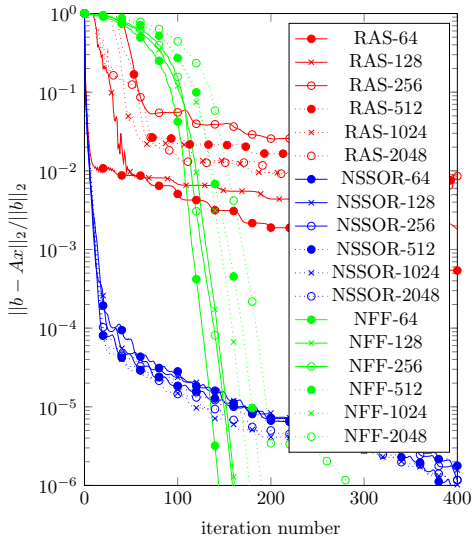

Figure 8: Convergence for `3DSKY225x225x225`.

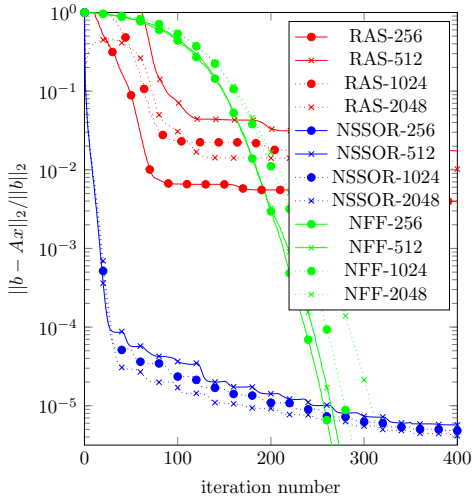

Figure 9: Convergence for `3DSKY400x400x400`.

best convergence rate. For `3DPOI225x225x225`, although NSSOR converges well, with the help of filtering techniques, NFF provides even a faster convergence rate. Furthermore, the iteration number of GMRES preconditioned by NFF increases slightly when increasing the number of subdomains. This is a very important feature to ensure the scalability of NFF. For `3DSKY225x225x225`, NFF has a slow convergence at the beginning, but catches up RAS and NSSOR. Both RAS and NSSOR get a faster convergence at the beginning, but stagnate later. We also observe that in the case where we use 2048 partitions, a restart causes another stagnation. Improvements, including filtering on more than one vector using low rank approximations, may solve this problem.
We also test two bigger matrices, `3DPOI400x400x400` and `3DSKY400x400x400`, which have around 64 million unknowns and 447 million nonzero entries. These tests start from 256 partitions to avoid running out of memory during the factorization of the diagonal blocks. Restart in GMRES is set to 350 to avoid stagnation of NFF. As shown in Figures 7 and 9, NFF converges always the fastest.

For `3DSKY225x225x225`, the time necessary for the construction and the application of the preconditioner as well as its breakdown into the different stages involved is given in Tables 1 and 2. Since the construction time depends only on the sparsity pattern of the input matrix, and does not depend on the matrix coefficients, the times given in these tables are thus valid for both matrices as far as the construction of the preconditioner is concerned. Table 1 displays the time used in different steps during the construction and the solving process of NFF where each innermost diagonal block is factored and solved using PARDISO. In this experiment, there is one MPI process per subdomain, and each MPI process is launched with 2 threads. In the construction part, by following the description given in Algorithm 1, columns $max(T_2^{i,j})$, $max(S_2^i)$ refer to $max_{i=1..q_1,j=1..q_2} fact(T_2^{i,j})$ and $max_{i=1}^{q1} fact(S_2^i)$ respectively, and display the maximum time required for the factorization of the subdomains and of the lower level separators respectively. The column (S_1^1) dis-

Table 1: Elapsed time in seconds for NFF on `3DSKY225x225x225` using PARDISO.

#cores	Partition	Compute NFF preconditioner						Apply NFF preconditioner (per iter)				
	$q_2 \times q_1$	$\max(T_2^{i,j})$	$\max(S_2^i)$	(S_1^1)	Reduce	max $L\beta U$	total	$\max(T_2^{i,j})$	$\max(S_2^i)$	(S_1^1)	comm	total
64	8×4	135.02	0.45	0.66	0.16	0.09	139.20	3.39	0.05	0.03	0.005	3.46
128	8×8	30.26	0.25	1.16	0.22	0.06	32.89	1.14	0.02	0.04	0.005	1.21
256	16×8	5.89	0.38	1.16	0.32	0.05	8.69	0.38	0.04	0.04	0.01	0.53
512	16×16	2.49	0.22	2.24	0.70	0.39	6.65	0.13	0.02	0.08	0.02	0.39
1024	32×16	2.34	0.33	2.20	0.87	0.52	7.89	0.05	0.04	0.08	0.02	0.34
2048	32×32	4.49	0.19	4.02	4.42	1.32	16.40	0.01	0.02	0.13	0.02	0.35

Table 2: Elapsed time in seconds for NFF on `3DSKY225x225x225` using PARDISO for subdomains and MUMPS for separators.

#cores	Partition	Compute NFF preconditioner						Apply NFF preconditioner (per iter)				
	$q_2 \times q_1$	$\max(T_2^{i,j})$	$\max(S_2^i)$	(S_1^1)	Reduce	max $L\beta U$	total	$\max(T_2^{i,j})$	$\max(S_2^i)$	(S_1^1)	comm	total
64	8×4	127.51	0.72	1.36	0.19	0.11	132.52	3.39	0.04	0.02	0.005	3.47
128	8×8	29.65	0.42	1.95	0.23	0.09	33.75	1.21	0.0008	0.05	0.01	1.27
256	16×8	6.07	0.68	1.98	1.07	0.85	11.23	0.40	0.04	0.03	0.04	0.60
512	16×16	2.65	0.37	3.13	1.30	0.87	19.83	0.14	0.03	0.06	0.06	0.44
1024	32×16	2.30	0.54	3.20	1.16	0.36	10.24	0.05	0.03	0.05	0.09	0.38
2048	32×32	4.44	0.30	4.47	4.73	1.02	17.69	0.02	0.02	0.09	0.03	0.33

Table 3: Elapsed time in seconds for NFF on `3DSKY400x400x400` using PARDISO.

#cores	Partition	Compute NFF preconditioner						Apply NFF preconditioner (per iter)				
	$q_2 \times q_1$	$\max(T_2^{i,j})$	$\max(S_2^i)$	(S_1^1)	Reduce	max $L\beta U$	total	$\max(T_2^{i,j})$	$\max(S_2^i)$	(S_1^1)	comm	total
512	16×16	38.11	1.48	14.4	2.21	0.08	58.04	1.07	0.05	0.22	0.03	1.53
1024	32×16	10.84	2.16	14.6	3.22	1.15	33.71	0.44	0.08	0.22	0.08	0.97
2048	64×16	5.81	3.50	14.6	5.16	0.46	32.05	0.17	0.12	0.22	0.12	0.81
4096	128×16	7.05	5.64	14.7	9.47	0.88	48.12	0.08	0.18	0.22	0.06	0.76

Table 4: Elapsed time in seconds for NFF on `3DSKY400x400x400` using PARDISO for subdomains and MUMPS for separators.

#cores	Partition	Compute NFF preconditioner						Apply NFF preconditioner (per iter)				
	$q_2 \times q_1$	$\max(T_2^{i,j})$	$\max(S_2^i)$	(S_1^1)	Reduce	max $L\beta U$	total	$\max(T_2^{i,j})$	$\max(S_2^i)$	(S_1^1)	comm	total
512	16×16	38.02	2.24	14.4	2.95	0.53	65.1	1.07	0.06	0.17	0.03	1.48
1024	32×16	11.50	2.86	14.3	4.16	0.57	35.7	0.44	0.09	0.17	0.92	0.92
2048	64×16	4.83	3.71	14.5	6.20	0.50	36.7	0.16	0.12	0.17	0.11	0.75
4096	64×32	6.16	5.51	14.7	14.59	0.78	49.0	0.09	0.15	0.16	0.05	0.67

plays the time required for the factorization of the upper level separator. When increasing the number of subdomains, the size of the subdomains decreases, but the size of the separators increases, that is $max(T_2^{i,j})$ decreases but $max(S_2^i)$ and (S_1) increases. Column *Reduce* indicates the overall time used for communicating and computing $\tilde{S}_k^1$. Since more subdomains imply more matrices to accumulate, Reduce time increases when the number of subdomains increases. Column $\max(L\beta U)$ indicates the maximum time used for the local matrix-matrix multiplication among all processors. As L and U are sparse and β is a diagonal matrix, the time necessary for computing $L\beta U$ is small compared to the other operations, and it decreases when the number of subdomains increases, since the size of L and U is related to the subdomain size. Due to load imbalance, this time increases when 2048 cores are used. Column *total* displays the overall time elapsed during the construction of NFF. We observe good speedups up to 512 subdomains, which correspond to 512 MPI processes, 1024 cores. However we do not obtain any speedup when more than 512 subdomains are used. This is because the separator becomes larger, and the time spend in factoring and solving the separator becomes more important. Even if the direct solver can use more processors, we did not obtain better results with our settings.

In the solving part, columns $\max(T_2^{i,j})$, $\max(S_2^i)$, and (S_1^1) display the maximum time per iteration required for solving in a subdomain, a lower level separator, and the upper level separator respectively. It includes not only the resolution time used in the innermost diagonal blocks, but also the time spent in matrix-vector multiplication and other local computations. Column *comm* displays the runtime of `MPI_Bcast` and `MPI_Reduce` in each iteration, the two communication routines used in Algorithms 3 and 4. It becomes more important when the number of subdomains increases, since the separators become larger. Column *total* indicates the overall time spent in applying the preconditioner during each iteration. Table 2 displays the same runtimes when MUMPS is used for factoring and solving in parallel the separators, where 4 MPI processes are used for each lower level separator and 16 MPI processes are used for the upper level separator. We do not obtain important gains for these two test cases.

For `3DSKY400x400x400`, results of same tests are displayed in Tables 3 and 4. MUMPS in Table 4 uses 8 MPI processes for each lower level separator and 16 MPI processes for the upper level separator. Compared to Table 3, with MUMPS, the time per iteration is reduced by almost 10%.

In summary, for `3DSKY225x225x225`, the application of NFF remains scalable up to 2048 cores, when the best runtime of 0.33 seconds per iteration is obtained. For the matrix `3DSKY400x400x400`, the application of NFF remains scalable up to 4096 cores, with a best runtime of 0.67 seconds per iteration. However the construction of NFF is scalable only up to 1024 cores. The usage of a distributed memory solver

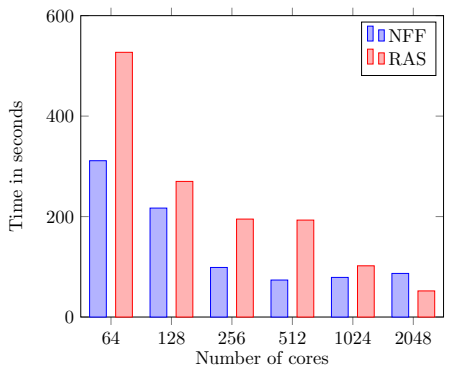

Figure 10: Comparison between NFF and RAS for `3DSKY225x225x225`, 2 threads are used per subdomain, 1 thread per core.

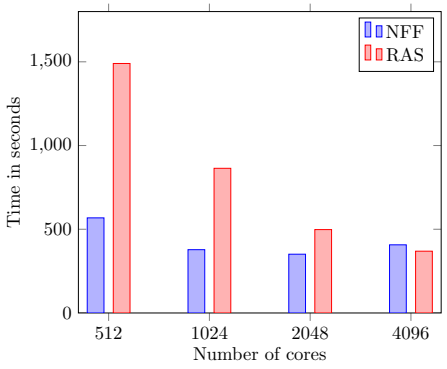

Figure 11: Comparison on `3DSKY400x400x400` between NFF and RAS, 2 threads are used per subdomain, 1 thread per core.

Table 5: Comparison on `3DSKY255x255x255` between NFF and RAS for a same relative error.

	RAS		NFF	
# subdomains	iteration	error	iteration	error
32	889	2.66e-07	50	2.69e-07
64	1639	3.00e-07	148	3.15e-07
128	2546	5.27e-07	162	7.94e-07
256	4664	6.20e-07	160	8.31e-07
512	4567	2.07e-07	191	7.28e-07
1024	3564	1.49e-07	179	5.82e-07

Table 6: Comparison on `3DSKY400x400x400` between NFF and RAS for a same relative error.

	RAS		NFF	
# subdomains	iteration	error	iteration	error
256	5485	5.90e-07	268	2.26e-06
512	6126	2.72e-06	273	3.20e-06
1024	7163	1.88e-06	289	2.60e-06
2048	10000	3.76e-06	317	3.80e-06

for factoring and solving the diagonal blocks corresponding to the separators becomes useful for matrices of very large size. Figures 10 and 11 present a comparison of the overall runtime of GMRES preconditioned by NFF and by RAS for both matrices. The overall runtime includes the construction of the preconditioner. On `3DSKY225x225x225` and `3DSKY400x400x400`, we let both methods to iterate until the relative error becomes smaller than 10^{-7} and 10^{-6} respectively. As the application of RAS requires just a local solve without any communication, the time of applying RAS per iteration is much smaller than that of NFF. However, to reach the same relative residual in the Krylov solver as NFF, RAS requires thousands of iterations, many more than NFF, as it can be seen in Tables 5 and 6. For `3DSKY225x225x225`, as displayed in Figure 10, for 64, 256, and 512 cores (where 2 threads are used per subdomain, 1 thread per core), GMRES preconditioned by NFF is more than twice faster than GMRES preconditioned by RAS. However with 1024 cores, NFF does not decrease any more its runtime, and hence it becomes slower than RAS with 2048 cores. For the larger `3DSKY400x400x400` matrix, NFF scales better. As shown in Figure 11, it scales up to 2048 cores. Figures 12 and 13 display the time elapsed in the construction and the application of NFF for the above experiment. As it can be seen, when the number of subdomains increases, the solving time is an important fraction of the overall runtime. This encourages us to develop a preconditioner which filters on more than one vector to further reduce the iteration number.

5. CONCLUSION

In this paper, we have described the data distribution and the parallel implementation of the nested filtering factorization preconditioner NFF. We showed through numerical experiments on matrices arising from the discretization of PDEs on three dimensional grids that the number of iterations of NFF increases slowly while increasing the number of processors. In terms of performance, NFF is scalable up to 2048 cores and it is up to 2.6 times faster than RAS. Our future work focuses on two main directions, improving the scalability of NFF by reducing the communication cost and applying a multilevel approach, and preserving more directions of interest.

The loss in scalability of NFF is mainly due to the increase of the size of the separators when the number of processors increases, which leads to an increase of the communication cost as well as of the factor and solve costs associated with the separators. To improve the scalability of NFF, we plan to use a multilevel approach in which the separators will be ordered into a nested block arrow structure and will be factored recursively. Also we plan to exploit the fact that the collectives used in the forward and backward solves are performed on sparse data, and this could further reduce the communication cost.

We will also focus on developing a preconditioner which preserves several directions of interest in order to get a better convergence rate. This will allow us to apply NFF on other challenging problems such as elasticity problems. Typically, the kernel of the elasticity system is made of rigid boly motions. Instead of filtering over one vector, we will filter simultaneously on three or six vectors depending on the dimension of the problem.

6. ACKNOWLEDGMENTS

This work was granted access to the HPC resources of TGCC @CEA made available within the Distributed European Computing Initiative by the PRACE-2IP, fund by the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement RI-283493. This

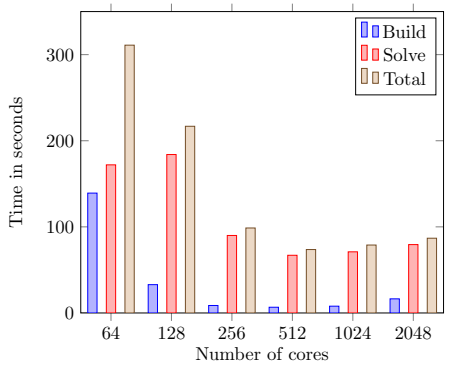

Figure 12: Time elapsed in the construction and the application of NFF on `3DSKY225x225x225`, 2 threads are used per subdomain, 1 thread per core.

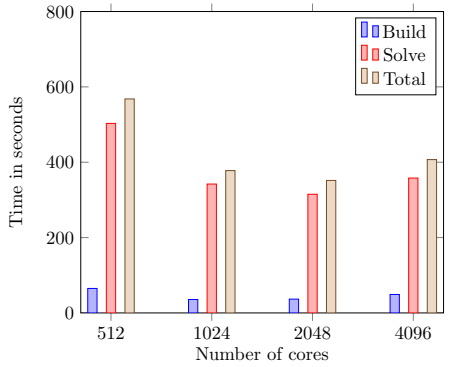

Figure 13: Time elapsed in the construction and the application of NFF on `3DSKY400x400x400`, 2 threads are used per subdomain, 1 thread per core.

work has been supported in part by the French National Research Agency (ANR) through COSINUS program, project PETALH no ANR-10-COSI-013.

7. REFERENCES

[1] Y. Achdou and F. Nataf. An iterated tangential filtering decomposition. *Numer. Linear Algebra Appl.*, 10(5-6):511–539, 2003. Preconditioning, 2001 (Tahoe City, CA).

[2] Y. Achdou and F. Nataf. Low frequency tangential filtering decomposition. *Numer. Linear Algebra Appl.*, 14(2):129–147, 2007.

[3] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[4] J. Appleyard and I. Cheshire. Nested factorization. In *Seventh SPE Symposium on Reservoir Simulation*, pages 315–324, 1983. paper number 12264.

[5] S. Balay, J. Brown, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2012. http://www.mcs.anl.gov/petsc.

[6] A. Brandt, J. Brannick, K. Kahl, and I. Livshits. Bootstrap AMG. *SIAM J. Sci. Comput.*, 33(2):612–632, 2011.

[7] I. S. Duff, M. A. Heroux, and R. Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum. *ACM Trans. Math. Softw.*, 28(2):239–267, June 2002.

[8] R. Fezzani, L. Grigori, F. Nataf, and K. Wang. Block Filtering Decomposition. In *Numerical Linear Algebra with Applications (NLAA)*, submitted 2012.

[9] J. A. George, J. W. H. Liu, and E. G. Ng. Communication results for parallel sparse Cholesky factorization on a hypercube. *Parallel Computing*, 10:287–298, 1989.

[10] L. Grigori, P. Kumar, F. Nataf, and K. Wang. A class of multilevel parallel preconditioning strategies. Research Report RR-7410, INRIA, Oct. 2010.

[11] L. Grigori and F. Nataf. Generalized Filtering Decomposition. Research Report RR-7569, INRIA, Mar. 2011.

[12] L. Grigori, F. Nataf, and P. Kumar. Multipurpose Calculation Computing Device, 2010. FR Patent WO/2012/035272 filed September 17, 2010, International Application No. PCT/FR2011/052128.

[13] L. Grigori, F. Nataf, and L. Qu. Overlapping for preconditioners based on incomplete factorizations and nested dissection. In *Numer. Linear Algebra Appl.*, submitted 2012.

[14] M. Gu, X. S. Li, and P. S. Vassilevski. Direction-preserving and Schur-monotonic semiseparable approximations of symmetric positive definite matrices. *SIAM J. Matrix Anal. Appl.*, 31(5):2650–2664, 2010.

[15] G. Karypis and V. Kumar. A software package for partitioning unstructured graphs, partitioning meshes and computing fill-reducing orderings of sparse matrices - verstion 4.0, 1998. See http://www-users.cs.umn.edu/karypis/metis.

[16] P. Kumar. *A class of preconditioning techniques suitable for partial differential equations on structured and unstructured mesh*. PhD thesis, University Paris-Sud XI, 2010. Inria Saclay - Ile de France.

[17] P. Kumar, K. Meerbergen, and D. Roose. Multi-threaded nested filtering factorization preconditioner. In P. Manninen and P. Oster, editors, *Applied Parallel and Scientific Computing*, volume 7782 of *Lecture Notes in Computer Science*, pages 220–234. Springer Berlin Heidelberg, 2013.

[18] Q. Niu, L. Grigori, P. Kumar, and F. Nataf. Modified tangential frequency filtering decomposition and its fourier analysis. *Numerische Mathematik*, 116(1):123–148, 2010.

[19] Y. Saad and M. H. Schultz. Gmres: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, July 1986.

[20] O. Schenk and K. Gartner. Solving unsymmetric sparse systems of linear equations with {PARDISO}. *Future Generation Computer Systems*, 20(3):475 – 487, 2004. Selected numerical algorithms.

[21] C. Wagner. Tangential frequency filtering decompositions for symmetric matrices. *Numer. Math.*, 78(1):119–142, 1997.